the *apply family

Andrea Gustafsen

The *apply family

lapply: loops over a list and applies a function to every element of that list (returns a list)
sapply: a variant of lapply that simplifies the results (returns a vector or matrix if possible)
apply: a function that loops over the margins (rows or columns) of an array, useful for taking summaries of matrices or higher dimensional arrays
tapply: short for "table apply". Applies a function over subsets of a vector

Sidebar: Anonymous functions: functions used inside the `*apply` functions

lapply

Arguments of the `lapply` function:

```
args(lapply)
```

```
## function (X, FUN, ...)
## NULL
```

`lapply` takes three arguments:

- `x` a list
- `FUN` a function
- `...` which can be used to pass arguments to the function

If `x` is not a list it will be coerced to a list and `lapply` always returns a list.

Example: Looping over a list with three elements, a, b and c and taking the mean of each.

```
# create a list of three vectors
my_list <- (list(a = 1:100, b = rnorm(100, 10, 4), c = rnorm(100, 20, 5)))

# taking the mean of each vector
sapply(my_list, mean)
```

```
##        a        b        c
## 50.50000 10.17261 21.10605
```

Example: R will loop over vector x and generate x number of random normal variables for each loop.

```r
# vector (1, 2, 3, 4, 5)
x <- 1:5

# generate a list with five elements, each containing x number of random normal variables
lapply(x, rnorm)
```

```
## [[1]]
## [1] -1.538374
##
## [[2]]
## [1] -1.5265030 -0.4929535
##
## [[3]]
## [1]  0.6240142 -0.4144008 -0.2726328
##
## [[4]]
## [1] -0.02952812  0.85179053  1.68338350  1.83450488
##
## [[5]]
## [1] -0.3908881 -0.7345581  0.6257804 -1.5280267 -0.3504460
```

Example cont.: We can specify the arguments of the function in FUN by passing them to the `...` argument.

```r
# generate numbers from the normal dist. with mean 10 and sd 2
x <- 1:5
lapply(x, rnorm, mean = 10, sd = 2) #default is mean = 0 and sd = 1
```

```
## [[1]]
## [1] 11.18355
##
## [[2]]
## [1] 11.311027  5.929414
##
## [[3]]
## [1]  9.085628  9.736445 11.310945
##
## [[4]]
## [1] 10.317975  6.045527 11.748884  7.557422
##
## [[5]]
## [1]  9.479017 10.571433  7.720449 12.612526  8.371037
```

Sidebar: Anonymous Functions

The `*apply` family functions make heavy use of anonymous functions. If we want to apply a function that does not already exist as a function in R, we need to write our own function directly within the apply function.

Example: Let us say we have two matrices and we want to extract a specific row or column, we can do this by passing an anonymous function.

```r
# create a list of two matrices
mat_list <- list(matrix(1:4, nrow = 2), matrix(1:9, nrow = 3))
mat_list
```

```
## [[1]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Here we extract the second row from each matrix by adding the anonymous function.

```r
lapply(mat_list, function(row2) row2[2,])
```

```
## [[1]]
## [1] 2 4
##
## [[2]]
## [1] 2 5 8
```

sapply

The only difference from `lapply` is that `sapply` always tries to simplify the result if possible. The output of the `lapply` function is always a list. The output in `sapply` will be:

- if the result is a list where each element is length 1: a vector
- if the result is a list where each element is a vector of same length > 1: a matrix
- otherwise a list

Example: result is a vector Taking the mean of each element in the list.

```r
#gives two elements of length 1, result as a vector
my_list <- (list(a = 1:10, b = 11:20, c = 21:30, d = 31:40))
sapply(my_list, mean)
```

```
##    a    b    c    d
##  5.5 15.5 25.5 35.5
```

Example: result is a matrix Extracting the first columns from each element in the list.

```r
# create three 2x2 matrices
mat3 <- list(matrix(1:4, 2, 2), matrix(5:8, 2, 2), matrix(9:12, 2, 2))

# using an anonymous function. sapply gives three vectors of equal length 2, result is a matrix
sapply(mat3, function(col1) col1[, 1])
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
```

apply

Used to evaluate a function over the **margins** of an array (most commonly applied to rows or columns of matrices or higher dimensional arrays).

```r
args(apply)
```

```
## function (X, MARGIN, FUN, ..., simplify = TRUE)
## NULL
```

The argument MARGIN specifies which margin that should be retained. 1 loops over the rows and 2 loops over the columns.

```r
# 10x5 matrix of random variables from the uniform dist between 1 and 10
my_uniform_mat <- matrix(runif(50, 1, 10), nrow = 10, ncol = 5)
my_uniform_mat
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]
##  [1,] 7.358656 4.954976 7.463178 8.746075 5.740126
##  [2,] 1.212831 7.802608 1.960541 7.331445 4.807338
##  [3,] 2.979665 7.056936 7.120395 2.225363 4.724028
##  [4,] 7.896394 9.084301 1.499743 3.419157 5.930389
##  [5,] 2.182960 5.146888 4.190333 6.412120 2.962805
##  [6,] 7.780662 5.987442 1.138422 2.324377 7.213856
##  [7,] 1.258458 5.651294 3.142168 6.053190 9.165259
##  [8,] 4.742305 6.349780 9.625302 8.017725 7.908878
##  [9,] 5.844137 5.178559 2.710158 1.461808 2.505471
## [10,] 9.952534 9.065870 5.978028 7.670140 4.519238
```

```r
#means for all rows
apply(my_uniform_mat, 1, mean)
```

```
##  [1] 6.852602 4.622953 4.821277 5.565997 4.179021 4.888952 5.054074 7.328798
##  [9] 3.540027 7.437162
```

```r
#means for all columns
apply(my_uniform_mat, 2, mean)
```

```
## [1] 5.120860 6.627865 4.482827 5.366140 5.547739
```

There are already functions for row/col sums and means in R:

- rowRums = apply(x, 1, sum)
- colSums = apply(x, 2, sum)
- rowMeans = apply(x, 1, mean)
- colMeans = apply(x, 2, mean)

These are optimized and more efficient than using the apply function. But we can evaluate other functions such as quantiles.

```
my_uni_mat <- matrix(runif(200, min= 0, max = 19), 20, 10)

# specifying which quantiles in the ... argument
apply(my_uni_mat, 1, quantile, probs = c(0.25, 0.75))
```

```
##          [,1]      [,2]      [,3]      [,4]      [,5]     [,6]      [,7]
## 25%  1.897038  2.816784  5.148095  5.365916  4.510524  5.23625  6.405452
## 75% 13.793826 17.653403 16.443517 15.756307 11.173076 14.02096 16.270027
##          [,8]     [,9]    [,10]    [,11]     [,12]     [,13]    [,14]    [,15]
## 25%  2.637544 2.831380 3.815013 12.35824  2.371836  4.858216  2.42734 3.935105
## 75% 10.488405 8.398546 9.588970 15.77627 11.874962 12.639145 11.38424 7.977802
##         [,16]     [,17]     [,18]     [,19]     [,20]
## 25%  5.515075  9.913019  7.057243  1.694762  3.120829
## 75% 12.691432 16.111116 13.806821 12.463269 16.210536
```

Example of a three dimensional array and collapsing one of the dimensions.

```
# think 10 2x2 matrices stacked together
x <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(x, MARGIN = c(1, 2), FUN = mean)
```

```
##            [,1]       [,2]
## [1,] -0.0658407 0.35360960
## [2,]  0.1620723 0.07845485
```

Same as

```
rowMeans(x, dim = 2)
```

```
##            [,1]       [,2]
## [1,] -0.0658407 0.35360960
## [2,]  0.1620723 0.07845485
```

tapply

`tapply` applies a function over a subsets of a vector. For this we need to specify a vector (arg INDEX) to identify which elements of the numeric vector we are going to calculate something on.

For example we can have a data set with men and women, and we want to evaluate the mean height for both groups respectively.

```r
args(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
## NULL
```

- X: a vector
- INDEX: a factor or a list of factors
- FUN: the function we want to apply
- … : other arguments to be passed to FUN

```r
# create a data frame
person <- c("Kalle", "Anders", "Karin", "Sigrid", "Herman", "Pelle", "Hanna", "Anna")
sex <- factor(c("Male", "Male", "Female", "Female", "Male", "Male", "Female", "Female"))
height <- c(195, 190, 155, 165, 178, 201, 169, 170)
nationality <- factor(c("Swedish", "Swedish", "Norwegian", "Danish", "Swedish", "Danish", "Swedish", "Da

my_df <- data.frame(person, sex, height, nationality)
my_df
```

```
##   person    sex height nationality
## 1  Kalle   Male    195     Swedish
## 2 Anders   Male    190     Swedish
## 3  Karin Female    155   Norwegian
## 4 Sigrid Female    165      Danish
## 5 Herman   Male    178     Swedish
## 6  Pelle   Male    201      Danish
## 7  Hanna Female    169     Swedish
## 8   Anna Female    170      Danish
```

```r
tapply(X = my_df$height, INDEX = sex, FUN = mean)
```

```
## Female   Male
## 164.75 191.00
```

Practical examples of using *apply

Calculate power of a one-sample t-test

The one sample t-test determines if an unknown population mean is different from a specific value.

Doing hypothesis testing, power is the probability of rejecting a false null hypothesis. In other words, it gives the probability that the test will detect a true difference in means.

Using simulation we can actually evaluate a test's power since we will know the true value of the parameter, in this case the mean.

Below we generate 1000 samples from the normal distribution with mean 1.75 and standard deviation sqrt(5/3). We test against null hypothesis that mean is 1 with significance level 95%. We do this for each of the 1000 samples by using the `apply` function, and calculate the proportion of times we reject the null hypothesis. We can see that the the test detects the difference 71% of the times.

```r
# function to generate data from the normal distribution
# S = number of samples
# n = sample size
# mu = mean
# sigma = standard deviation
generate.normal <- function(S,n,mu,sigma){
  # generate one data sat, each row is one sample
  data <- matrix(rnorm(n*S,mu,sigma), ncol = n, byrow = T)
}


# set seed and parameter values
# generate 1000 samples from the normal distribution
set.seed(28)
S <- 1000
n <- 20
sigma <- sqrt(5/3)
mu <- 1.75
mu0 <- 1 #test generated data against this mean

# generate data from normal
data <- generate.normal(S, n, mu, sigma)

# calculate test statistic for each sample
# using apply here to gets the mean and variance for each sample in the generated data
ttest_statistic <- (apply(data, 1, mean) - mu0) / sqrt(apply(data, 1, var) / n)

# rejection area in the t-distribution
# two sided test, alpha 0.05, n-1 degrees of freedom
t05 <- qt(0.975, n-1)

# calculate power of test (proportion of rejections)
power <- sum(abs(ttest_statistic)>t05)/S
power
```

```
## [1] 0.707
```

For this parameter (normal with mean 1.75 and standard deviation sqrt(5/3)), using sample size of 20 and testing that the mean is 1, we will detect the difference in 71% of the times.

How large sample size is needed for the test? (Run the function over several values of one parameter)

If we want to see how large sample size we need to get a desired power of at least 80 percent, we can make use of the `sapply` function to test several values of sample sizes in one call.

We put the code above in a function called `ttest_power` so that we can pass it to the `sapply`function.

```r
# put together into a function
ttest_power <- function(seed = 28, S=1000, n = 20, mu = 0, sigma = 1, mu0 = 0.1){
    set.seed(seed)
    # function to generate data from the normal distribution
```

```
  generate.normal <- function(S,n,mu,sigma){

    data <- matrix(rnorm(n*S,mu,sigma), ncol = n, byrow = T)
    return(data)
  }

  # generate data from normal
  data <- generate.normal(S, n, mu, sigma)

  # calculate test statistic
  ttest_statistic <- (apply(data, 1, mean) - mu0) / sqrt(apply(data, 1, var) / n)

  # rejection area in the t-distribution
  t05 <- qt(0.975, n-1)

  # calculate power of test
  power <- sum(abs(ttest_statistic)>t05)/S

  return(power)
}
```

We create a vector of different sample sizes that we will test in the function. We can manipulate any of the parameters in the `ttest_function` depending on what we wish to test. In this case we run the function with different values of sample size, `n`.

```
# vector with sample sizes we wish to evaluate
sample_size <- c(10, 15, 20, 25, 30, 35, 40)

# use sapply to test different sample sizes in the ttest_power function
power <- sapply(sample_size, function(x)ttest_power(S = 1000, n = x, mu = 0, sigma = 1, mu0 = 0.5))

#add sample sizes as names to the result
names(power) <- sample_size
power
```

```
##    10    15    20    25    30    35    40
## 0.311 0.448 0.563 0.676 0.748 0.820 0.868
```

If we have data with mean 0 and standard deviation 1, and test against a null hypothesis mean of 0.5, we must have a sample size of at least 35 to obtain power higher than 80%. We can see that if our sample size is 20, the probability of detecting the true difference is only 56%.

Produce several plots

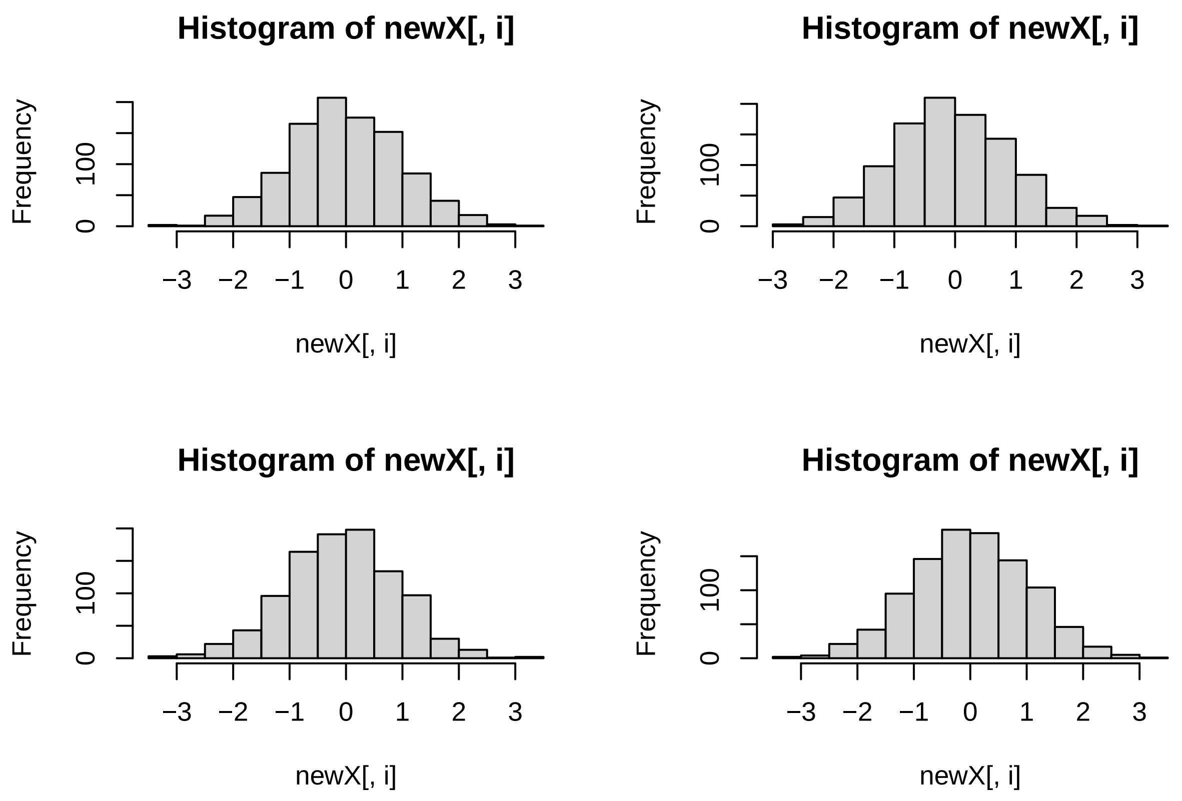
`apply` can also be used to produce several plots in one call. If we want to view the distribution of some generated data we can do this using apply.

```
# generate four samples of 1000 random variable from the normal distribution
random_normal <- generate.normal(S = 4, n = 1000, mu = 0, sigma = 1)

# display plots 2 by 2
```

```r
par(mfrow=c(2,2))

# samples are stored row wise, so we call margin = 1
apply(random_normal, 1, hist)
```



So these are just a few examples of how the `*apply` family can make plenty of calculations in just a couple of lines of code! I really find the `*apply`-functions Super-Duper useful!